МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Алгоритмы и структуры данных»

Тема: Программирование алгоритмов с бинарными деревьями

Вариант 8в

Студент гр. 8304 _____ Масалыкин Д. Р.
Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2019

Цель работы.

Познакомиться с такой часто используемой на практике, особенно при решении задач кодирования и поиска, нелинейной структурой данных, как бинарное дерево, способами её представления и реализации, получить навыки решения задач обработки бинарных деревьев.

Постановка задачи.

Вариант 1в: Рассматриваются бинарные деревья с элементами типа Elem (в качестве Elem использовать char). Заданы перечисления узлов некоторого дерева b в порядке КЛП. Требуется:

а) восстановить дерево b и вывести его изображение;

б) перечислить узлы дерева b в порядке ЛПК.

Описание алгоритма.

Для начала программа должна считать данные и занести их в структуру. Дерево реализуется на базе вектора: в структуре каждого узла должен храниться индекс левого и правого узлов в массиве. Если узел в дереве пустой, то массив с данным индексом хранит нулевой элемент.

Для отображения дерева необходимо идти от корня к листьям при помощи КЛП-обхода и отображать каждый узел с соответствующим отступом, который увеличивается на каждом шаге рекурсии. Для отображения в формате ЛПК рекурсивно обходится дерево и при условии отсутствия потомков печатается узел. Если потомки есть, вызывается эта же функция для левого и правого потомка, а потом печатается сам элемент.

Спецификация программы.

Программа предназначена для определения вхождения элемента в дерево, подсчета количества искомых элементов, нахождение длины пути до ближайшего элемента.

Программа написана на языке C++. Входными данными является дерево в упрощенной скобочной записи. Данные вводятся либо из файла, либо с помощью GUI. Выходными данными является конечный результат. Данные выводятся на экран монитора.

Тестирование.

Результаты тестирования программы приведены в табл. 1.

Таблица 1 – Результаты тестирования

Ввод	Вывод
(a(b)(c))	Your tree list form: a b c LPK:((b)(c)a)
dfhakdjshfjakshfjakshfjkashf	INCORRECT STRING!!!!!

Больше тестов можно увидеть запустив скрипт для тестирования.

Описание функций и СД.

```
void print_tree(MyBinaryTree<char> tree, std::string space, int index);
```

Функция принимает на вход само дерево, строку с отступом для визуализации, а также индекс текущего элемента. Для печати Функция вызывается рекурсивно для каждого поддерева.

```
void print_LPK(MyBinaryTree<char> tree, int index);
```

Функция принимает дерево и индекс текущего элемента. Для печати функция последовательно рекурсивно вызывается для левого поддерева и правого поддерева, а после этого печатается узел.

Выводы.

В ходе выполнения данной лабораторной работы был реализован шаблонный класс бинарное дерево на блазе вектора, а также написаны функции поиска для работы с ним.

Приложение А. Исходный код программы

main.cpp

```
#include <iostream>
#include <string>
#include <fstream>
#include "MyBinaryTree.h"

void print_tree(MyBinaryTree<char> tree, std::string space, int index);
void print_LPK(MyBinaryTree<char> tree, int index);
int check_str(std::string str);

int main(int argc, char* argv[]) {
    std::string inp_str;
    if(argc == 1){
        std::cout<<"No input file! Enter your tree with keyboard!"<<std::endl;
        std::cin>>inp_str;
    }else {
        std::ifstream file;
        file.open(argv[1]);
        if (!file.is_open()) {
            std::cout << "Can't open file" << std::endl;
            return 1;
        } else {
            getline(file, inp_str);
        }
    }
    if(!check_str(inp_str)){
        std::cout<<"INCORRECT STRING!!!!!"<<std::endl;
        return 1;
    }
    MyBinaryTree<char> MyTree;
    MyTree.createCharTree(inp_str);
    std::cout<<MyTree.toStdString()<<std::endl;

    std::cout<<"Your tree list form:"<<std::endl;
    std::string space = "\t";
    std::cout<<MyTree.getElem(0)<<std::endl;
    print_tree(MyTree, space, 0);
    std::cout<<"LPK:(";
    print_LPK(MyTree, 0);
    std::cout<<')'<<std::endl;

    return 0;
}

void print_tree(MyBinaryTree<char> MyTree, std::string space, int index){
    if(MyTree.getLeft(index) != -1) {
        std::cout << space << MyTree.getElem(MyTree.getLeft(index)) << std::endl;
        print_tree(MyTree, space + '\t', MyTree.getLeft(index));
    }
    if(MyTree.getRight(index) != -1) {
        std::cout << space << MyTree.getElem(MyTree.getRight(index)) << std::endl;
        print_tree(MyTree, space + '\t', MyTree.getRight(index));
    }
}


void print_LPK(MyBinaryTree<char> tree, int index){
    if(tree.getLeft(index) == -1 && tree.getRight(index) == -1){
        std::cout<<tree.getElem(index);
    }
    else if(tree.getLeft(index) != -1 && tree.getRight(index) == -1){
        std::cout<<'(';
        print_LPK(tree, tree.getLeft(index));
        std::cout<<')';
        std::cout<<tree.getElem(index);
    }
    else if(tree.getLeft(index) == -1 && tree.getRight(index) != -1){
        std::cout<<'(';
        print_LPK(tree, tree.getRight(index));
        std::cout<<')';
        std::cout<<tree.getElem(index);
    }
    else if(tree.getLeft(index) != -1 && tree.getRight(index) != -1){
        std::cout<<'(';
        print_LPK(tree, tree.getLeft(index));
```

```
        std::cout<<')';
        std::cout<<'(';
        print_LPK(tree, tree.getRight(index));
        std::cout<<')';
        std::cout<<tree.getElem(index);
    }
}

int check_str(std::string str){
    int br_cntr = 0;
    bool is_br = false;
    for(int i = 0; i < str.size(); i++){
        if(str[i] == '(') {
            br_cntr++;
            is_br = true;
        }
        if(str[i] == ')') {
            br_cntr--;
            is_br = true;
        }
    }
    if(!is_br)
        return 0;
    if (br_cntr)
        return 0;
    else
        return 1;
}
```

mybinarytree.h

```
//
// Created by Dan on 26.11.2019.
//

#ifndef MYBINARYTREE_H
#define MYBINARYTREE_H

#include <iostream>
#include <ostream>
#include <memory>


constexpr size_t SIZE = 500;

template <class T>
class MyBinaryTree
{
    /* Класс-реализация бинарного дерева на массиве.
     * Содержит массив элементов дерева и размер дерева.
    */

public:
#pragma pack(2)
    //Структура элемента дерева
    struct Node
    {
        int leftElem;
        int rightElem;
        bool isNull;
        T data;
    };
    typedef std::shared_ptr<Node> NodeP;
#pragma pack()

protected:
    NodeP binTree[SIZE];
    size_t size;

public:
    explicit MyBinaryTree();
    ~MyBinaryTree() = default;
    void clearTree();

    MyBinaryTree& operator=(const MyBinaryTree& tree) = delete;
    //MyBinaryTree(const MyBinaryTree&) = delete;

    //Getters-методы
```

```
    bool isNull(size_t index) const;
    int getLeft(size_t index) const;
    int getRight(size_t index) const;
    size_t getSize() const;
    const T& getElem(size_t index) const;

    //Методы для решения подзадач. Шаблон метода для создания дерева реализован только для
char
    size_t createCharTree(const std::string& expression) = delete;
    int minHeightToElem(const T& elem) const;
    bool isExistElem(const T& elem) const;
    size_t countElem(const T& elem) const;

    std::string toStdString() const;
    std::string toAssignmentList() const;

protected:
    //метод для создания элемента
    void createNode();
    //метод для преобразования элемента к строке
    void NodeToStdString(const MyBinaryTree* tree, NodeP subTree, std::string& str) const;
    //метод для получения индекса в строке, где заканчивается поддерево
    size_t getEndIndexSubTree(const std::string& str, size_t indexEnd) const;
    //метод для преобразования дерева к уступчатому списку
    void _toAssignmentList(std::string& str, size_t index, size_t depth = 1) const;

    //приватные методы для подзадач, принимают на вход доп.параметры
    int _minHeightToElem(const T& elem, size_t heght = 0,
                         size_t index = 0, size_t depth = 1) const;
    bool _isExistElem(const T& elem, size_t index = 0, size_t depth = 1) const;
    size_t _countElem(const T& elem, size_t index = 0, size_t depth = 1) const;
};



template<class T>
MyBinaryTree<T>::MyBinaryTree()
{
    /*
     * Создание пустого дерева
    */

    this->size = 0;
}



template<typename T>
void MyBinaryTree<T>::clearTree()
{
    /*
     * Полное удаление дерева
    */

    for (size_t i = 0; i < size; ++i) {
        binTree[i].reset();
    }
    size = 0;
}



template <class T>
void MyBinaryTree<T>::createNode()
{
    /*
     * Создание элемента дерева.
     * Метод приватный т.к создание элементов должно происходить при создании дерева,
     * из-за особенностей реализации на массиве
    */

    NodeP tmp = NodeP(new Node);
    tmp->isNull = true;
    tmp->leftElem = -1;
    tmp->rightElem = -1;

    this->binTree[size] = tmp;
    size += 1;
}
```

```
template<typename T>
bool MyBinaryTree<T>::isNull(size_t index) const
{
    /*
     * Метод возвращает bool-значение, является ли элемент дерева нулевым
    */

    if (index < size) {
        return binTree[index]->isNull;
    }
    else {
        return false;
    }
}


template<class T>
int MyBinaryTree<T>::getLeft(size_t index) const
{
    /*
     * Метод возвращает индекс левого поддерева
    */

    if (!this->isNull(index)) {
        return this->binTree[index]->leftElem;
    }
    else {
        return -1;
    }
}


template<class T>
int MyBinaryTree<T>::getRight(size_t index) const
{
    /*
     * Метод возвращает индекс правого поддерева
    */

    if (!this->isNull(index)) {
        return binTree[index]->rightElem;
    }
    else {
        return -1;
    }
}


template<class T>
size_t MyBinaryTree<T>::getSize() const
{
    /*
     * Метод возвращает размер дерева
     */

    return this->size;
}


template<class T>
const T& MyBinaryTree<T>::getElem(size_t index) const
{
    /*
     * Метод возвращает значение элемента по индексу
    */

    if (!this->isNull(index)) {
        return this->binTree[index]->data;
    }
    else {
        exit(1);
    }
}
```

```
template <> inline
size_t MyBinaryTree<char>::createCharTree(const std::string& str)
{
    /*
     * Метод создания дерева. Строка должна быть корректна
     */

    size_t sizeRoot = this->size;
    this->createNode();
    char elem = 0;

    size_t indexStart = 1;
    while (str[indexStart] != '(' && str[indexStart] != ')') {
        if (str[indexStart] != ' ' && str[indexStart] != '(' && str[indexStart] != ')') {
            elem = str[indexStart];
            this->binTree[sizeRoot]->data = elem;
            this->binTree[sizeRoot]->isNull = false;
        }
        indexStart++;
    }

    if (str[indexStart] == ')') {
        return sizeRoot;
    }

    size_t indexEnd = indexStart + 1;
    indexEnd = getEndIndexSubTree(str, indexEnd);
    size_t leftTree = createCharTree(str.substr(indexStart, indexEnd - indexStart));
    this->binTree[sizeRoot]->leftElem = static_cast<int>(leftTree);

    indexStart = indexEnd;
    while (str[indexStart] != '(' && str[indexStart] != ')') {
        indexStart++;
    }

    if (str[indexStart] == ')') {
        return sizeRoot;
    }

    indexEnd = indexStart + 1;
    indexEnd = getEndIndexSubTree(str, indexEnd);
    size_t rightTree = createCharTree(str.substr(indexStart, indexEnd - indexStart));

    this->binTree[sizeRoot]->rightElem = static_cast<int>(rightTree);
    return sizeRoot;
}


template<class T>
size_t MyBinaryTree<T>::getEndIndexSubTree(const std::string &str, size_t indexEnd) const
{
    /*
     * Метод возвращает индекс в строке конца поддерева
    */

    size_t openB = 1;
    size_t closeB = 0;
    while (openB != closeB) {
        if (str[indexEnd] == '(') {
            openB++;
        }
        else if (str[indexEnd] == ')') {
            closeB++;
        }
        indexEnd++;
    }
    return indexEnd;
}


template<class T>
void MyBinaryTree<T>::_toAssignmentList(std::string &str, size_t index, size_t depth) const
{
    for (size_t i = 0; i < depth; ++i) {
```

```
        str += " ";
    }
    str += getElem(index);
    str += "\n";

    if (getLeft(index) != -1) {
        _toAssignmentList(str, getLeft(index), depth + 1);
        if (getRight(index) != -1) {
            _toAssignmentList(str, getRight(index), depth + 1);
        }
    }
}



template<class T>
std::string MyBinaryTree<T>::toStdString() const
{
    /*
     * Метод возвращает представление бинарного дерева в виде упрощенной скобочной записи
     * Элемент должен иметь перегрузку опрератора += std::string
    */

    if (size == 0)
        return "()";

    std::string tree;
    tree += "(";
    tree += this->binTree[0]->data;
    if (this->binTree[0]->leftElem != -1) {
        NodeToStdString(this, this->binTree[this->binTree[0]->leftElem], tree);
    }

    if (this->binTree[0]->rightElem != -1) {
        NodeToStdString(this, this->binTree[this->binTree[0]->rightElem], tree);
    }
    tree += ")";
    return tree;
}



template<class T>
std::string MyBinaryTree<T>::toAssignmentList() const
{
    if (size == 0) {
        return "(empty)";
    }

    std::string str;
    str += getElem(0);
    str += "\n";
    if (getLeft(0) != -1)
        _toAssignmentList(str, getLeft(0));
    if (getRight(0) != -1)
        _toAssignmentList(str, getRight(0));

    return str;
}



template<class T>
void MyBinaryTree<T>::NodeToStdString(const MyBinaryTree* tree, MyBinaryTree::NodeP
subTree, std::string &str) const
{
    /*
     * Метода записывает в строку-результат представление элемента дерева в упрощенной
     * скобочной записи
    */

    if (!subTree->isNull) {
        str += "(";
        str += subTree->data;
        if (subTree->leftElem != -1) {
            NodeToStdString(tree, tree->binTree[subTree->leftElem], str);
        }

        if (subTree->rightElem != -1) {
```

```
            NodeToStdString(tree, tree->binTree[subTree->rightElem], str);
        }
        str += ')';
    }
    else {
        str += "()";
    }
}


template<class T>
bool MyBinaryTree<T>::isExistElem(const T& elem) const
{
    /*
     * Метод для определения существования элемента в дереве, возвращает bool
    */

    if (size < 1) {
        return false;
    }
    else {
        return _isExistElem(elem);
    }
}


template<class T>
size_t MyBinaryTree<T>::countElem(const T &elem) const
{
    /*
    * Метод для подсчета количества вхождение элемента в дереве, возвращает size_t
   */

    if (size < 1) {
        return 0;
    }
    else {
        return _countElem(elem);
    }
}


template<typename T>
int MyBinaryTree<T>::minHeightToElem(const T &elem) const
{
    /*
     * Метод для определения минимального пути до
     * элемента в дереве, возвращает -1, если элемента в дереве нет
    */
    if (size < 1) {
        return -1;
    }
    else {
        return _minHeightToElem(elem);
    }
}


template<typename T>
bool MyBinaryTree<T>::_isExistElem(const T &elem, size_t index, size_t depth) const
{
    std::string dbgStr;
    for (size_t i = 0; i < depth; ++i) {
        dbgStr += " ";
    }

    bool isLeftTreeExistElem = false;
    bool isRightTreeExistElem = false;
    if (this->isNull(index)) {
        return false;
    }
    else if (index < this->size) {
        if (this->getElem(index) == elem)
            return true;

        if (this->getLeft(index) != -1)
```

```
            isLeftTreeExistElem = this->_isExistElem(elem, this->getLeft(index), depth +
1);

        if(this->getRight(index) != -1)
            isRightTreeExistElem= this->_isExistElem(elem, this->getRight(index), depth +
1);

        return isLeftTreeExistElem || isRightTreeExistElem;
    }
    else {
        return false;
    }
}


template<typename T>
size_t MyBinaryTree<T>::_countElem(const T &elem, size_t index, size_t depth) const
{
    std::string dbgStr;
    for (size_t i = 0; i < depth; ++i) {
        dbgStr += " ";
    }

    size_t count = 0;
    if (this->isNull(index)) {
        return 0;
    }
    else if (index < this->size) {
        if (this->getElem(index) == elem)
            count += 1;

        if (this->getLeft(index) != -1)
            count += this->_countElem(elem, this->getLeft(index), depth + 1);

        if(this->getRight(index) != -1)
            count += this->_countElem(elem, this->getRight(index), depth + 1);

        return count;
    }
    else {
        return 0;
    }
}



template<typename T>
int MyBinaryTree<T>::_minHeightToElem(const T &elem, size_t height,
                                      size_t index, size_t depth) const
{
    std::string dbgStr;
    for (size_t i = 0; i < depth; ++i) {
        dbgStr += " ";
    }

    if (isNull(index)) {
        return -1;
    }
    if (binTree[index]->data == elem) {
        return static_cast<int>(height);
    }

    int leftIndex = binTree[index]->leftElem;
    int rightIndex = binTree[index]->rightElem;

    int minHeightLeft = -1;
    int minHeightRight = -1;

    if (leftIndex != -1) {
        minHeightLeft = _minHeightToElem(elem, height + 1, leftIndex, depth + 1);
    }

    if (rightIndex != -1) {
        minHeightRight = _minHeightToElem(elem, height + 1, rightIndex, depth + 1);
    }
```

```
    if (minHeightLeft == -1 && minHeightRight == -1) {
        return -1;
    }
    else if (minHeightLeft != -1 && minHeightRight == -1) {
        return minHeightLeft;
    }
    else if (minHeightRight != -1 && minHeightLeft == -1) {
        return minHeightRight;
    }
    else {
        return std::min(minHeightLeft, minHeightRight);
    }
}

#endif // MYBINARYTREE_H
```